

PROGRAMMING EXPERT

Introducing .NET 3.0 and WPF

We say farewell to current Windows programming as **Mike James** casts off his programming shackles and embraces the Windows Presentation Foundation



FAQ

Q Please can you tell me what an API is, and what does it do?

A API stands for application programming interface. In the early days of software, companies would provide programmers with details of how to extend their products. This would take the form of documentation explaining what subroutines and functions the application made available for other programs to use. In time, API documentation became supplemented by example programs and templates that made working with the API easier. Such a package became known as a software development kit (SDK).

Today, the API is an object hierarchy – a set of objects, methods and properties available for programmers to use. But what you get when you download an SDK or API can be of variable quality.

Mike James

IT author, lecturer and programmer

mike@computershopper.co.uk

If you're comfortable with the way Windows programs work, the bad news is it's all about to change. The Windows API, which we use to create forms, buttons and whole user interfaces, is about to be replaced by a system based on the managed code of .NET 3.0. This is a revolution in Windows programming and should result in more impressive and easier-to-use programs. Currently the system is undergoing beta testing, but it is close to being released so it is worth finding out how it works and what you can do with it.

If you've been following the progress of Windows Vista, you've probably heard the codename Avalon. This is the part of .NET 3.0 we're interested in. A newer name for it is the Windows Presentation Foundation, or WPF. This is easy, powerful and fun, and is responsible for Windows Vista's new look. You'll be able to create .NET 3.0 programs for Windows 2003 and XP as well as Vista, but not earlier versions of Windows. If you have Windows Me, 2000, 98 or below, it's time to upgrade.

GETTING STARTED

To get going with .NET 3.0 you need to have at least C# Express or Visual Studio 2005 installed on Windows XP with SP2, or the latest beta of Vista. It will also work with Visual Basic and Visual Basic Express, and there is very little to change to make the examples work. You must download and install three additional modules: the .NET Framework 3.0 Runtime Components, the Windows SDK for Vista and .NET 3.0 runtime, and the strangely named Orcas .NET Framework 3.0 Development Tools.

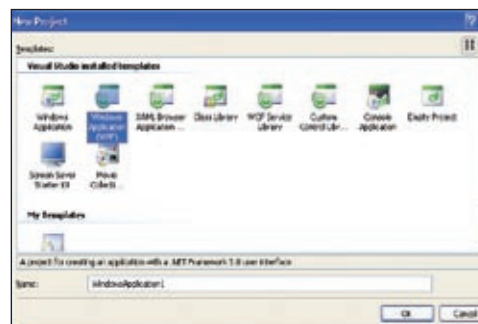
Each of these brings something extra to your current system. The .NET 3.0 runtime components allow you to run .NET 3.0 applications on Windows XP. If you're using a Vista beta, you don't need this as it's a standard part of the operating system. The SDK has all the documentation and examples. The Orcas extensions add new project types to Visual Studio or C# Express so

you can develop .NET 3.0 applications including WPF applications. Orcas will be a major part of the next version of Visual Studio.

It's important your three downloads are of the same vintage. There have been problems getting things working with downloads of different ages. Download and use the latest version of each component from the Microsoft website; simply search for .NET 3.0 and check the Downloads link. This changes frequently, so we won't quote the current URL. If you've installed earlier versions of any of the modules, remove them before installing the new ones. You shouldn't use any of this software on a PC you're using for anything important, though the current version seems reasonably stable.

NEW PROJECTS

After you have installed the necessary software, you'll see there are new project types in Visual Studio: Windows Application (WPF), XAML Browser Application, WCF Service Library and Custom Control Library (WPF). The WPF project is the new way to create applications in .NET 3.0 and will eventually replace the old familiar Windows Application.



▲ The new .NET 3.0 projects are added to the old

If you open a new Windows Application (WPF) project, you will discover many of the immediately obvious differences between WPF and Windows Forms. The most obvious is that there is a new file type called Window1.xaml by default. The Extensible Application Markup Language (XAML), pronounced 'zamel', defines user interfaces without the need for programming. You could think of it as HTML on steroids, but there is much more to it than that.

XAML is probably the single biggest innovation in WPF. As well as the .xaml file, the system generates an associated .cs code file with the same name. This is where you write the code that manipulates the user interface components specified in the XAML file. If you've ever worked with ASP.NET, this will sound familiar; it is HTML with code behind. The Orcas extension adds Intellisense prompting for WPF and for XAML in particular. As you write code, Visual Studio will suggest the correct syntax and relevant methods and properties. However, this isn't currently foolproof and occasionally it doesn't offer you valid choices. Often it doesn't give you any options at all.

If you run the default project, it should compile successfully and display a blank form. This is reassuring, but not very exciting. To make it a little more interesting, let's add a button and make it do something. The principles behind our first program will be explained in a moment, but you should be struck by how similar it all is to using HTML to create a webpage. Select the XAML file's tab and between the <Grid> and </Grid> tags add the following:

```
<Button
    Name="Button1"
    Height="50"
    Width="200"
    Click="OnClick1" >
    Click Me
</Button>
```

This is the XAML markup for a button called Button1. We've specified the size but not the location, so it appears at the centre of the form. A click event handler is also defined as OnClick1 and the initial content (the button's label) is set to Click Me. Select the code page and enter:

```
void OnClick1(object sender,
    RoutedEventArgs e)
{
    Button1.Content = "Hello World";
}
```



▲ Your first WPF application

As you type, you'll see Visual Studio knows the name of the button you have defined using XAML, and it automatically makes the connection between the OnClick function and the event handler you specified in the XAML. If you run the program, you will see a button in the middle of the form inviting you to click it. When you do, it will say Hello World.

WHY WPF?

At this point, you may be wondering what all the fuss is about. After all, it's easy enough to create a form with a button without WPF. There are lots of reasons why WPF is the way forward, though. It uses the modern graphics hardware that nearly all computers have. The Windows Forms API was invented in the days when graphics cards were simple and it makes very little use of hardware graphics acceleration. WPF uses the DirectX 9 rendering engine and this taps the power of your graphics card. It also means WPF has access to a wide range of 3D visual effects including meshes, materials, lights, textures and so on. WPF provides you with yet another way of creating 3D graphics under Windows.

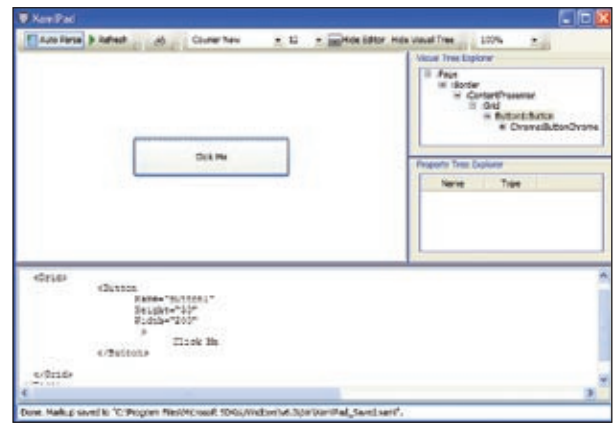
At a more mundane level, the elements of any UI you create using WPF are rendered using vector graphics rather than bitmaps. This means they display at the highest resolution the display device supports. It makes the user interface more device-independent than the old way of working, which often resulted in silly sizes or arrangements of controls if the user was working at a different resolution than expected. It is also a retained mode graphics system, which makes anything you draw on a form persistent without you having to redraw it. If you draw a line on a form and the form is minimised, the line will still be there when the form is restored; you don't have to redraw it. Retained mode allows the graphics system to optimise the display process.

XAML can be used to create web applications. In fact, there are two distinct types of WPF application: Navigation Window and Document Window. A Navigation Window provides hyperlinks to allow the user to move through the application, whereas a Document Window is more like a traditional application. WPF can also be used on reduced hardware and mobile devices, and it is destined to be the only way of creating user interfaces and graphics under all versions of Windows. In short, WPF is the way you would implement Windows Forms if you were starting with today's technology.

HOW WPF WORKS

So has Windows changed to be a webpage-like system? Not really. If you don't want to use XAML you don't have to. There are still .NET classes for every WPF component. For example, as well as a <Button> tag there is a Button class within the framework, not to be confused with the Windows Forms Button class. If you want to code our first example using minimal XAML, delete the grid tags and everything between them and then enter the following code:

```
public Button Button1;
```



▲ XamlPad in action

```
public Window1()
{
    Grid g = new Grid();
    Button1 = new Button();
    Button1.Height = 50;
    Button1.Width = 200;
    Button1.Content = "Click Me";
    Button1.Click += OnClick1;
    g.Children.Add(Button1);
    this.Content = g;
    InitializeComponent();
}

void OnClick1(object sender,
    RoutedEventArgs e)
{
    Button1.Content = "Hello World";
}
```

It looks more complicated, and it's longer, but you should be able to see it does the same job as the XAML by creating objects and setting their properties. When you use XAML to define an interface, the system converts the XAML tags into the corresponding objects and property settings for you. So whether you use XAML or not, the result is the same. In this article, the focus will be on using XAML, but converting the examples into code is easy.

Before long you won't have to write XAML the hard way. A form designer is under development, codenamed Cider, but it currently works only with Visual Studio and hasn't got far. If you want help creating XAML files, there's XamlPad, which is included with the SDK. With XamlPad, you enter XAML code, it is checked for correctness and the resulting form is displayed; it isn't a drag-and-drop designer. Even when one is available, you still need to know how to write XAML (or the equivalent code) to do things that go beyond the basics.

LAYOUT AND CONTROLS

There are lots of new features in WPF to explore, but before we do anything exciting we'll look at the way a user interface is put together.

WPF has all the user interface controls you would expect – buttons, text boxes, drop-down lists – and many more advanced controls such as calendars and menu bars. However, at the top of it all is the Visual class from which any visible element is descended. Every visual component has a Content property, which roughly corresponds to what is displayed. A Window object has a Content property, which specifies what it displays. In some cases, all you need to do is set a Content property to an appropriate value. For example, Button1.Content= "Click

O'REILLY

JavaScript: The Definitive Guide

★★★★★

£21



JavaScript is often dismissed as just another scripting language, but it is actually a powerful and complex language. Author David Flanagan does it justice in describing its strong points, its subtleties and its straightforward logic. Sometimes the book's efforts to cover absolutely everything are irritating, but as Javascript is an under-documented language it deserves this comprehensive treatment.

Flanagan does a good job of demystifying JavaScript, even if he is obsessive in places. The coverage of Ajax methods is a bit thin, and a large reference section is mostly a waste of paper. Overall, however, this book is well worth buying if you want to take JavaScript seriously.

SUMMARY

☒ Excellent, detailed guide

☒ Not much on Ajax

SUPPLIER www.amazon.co.uk
ISBN 0596000480
PAGES 1,018

DEITEL

Visual Basic 2005 for Programmers

★★

£28



I despair whenever I embark on a book by Harvey and Paul Deitel. They cover standard stuff in great detail and provide lots of long, complex examples. This particular attempt is typical. It's full of minute explanation of the obvious and rarely rises above what you could work out for yourself. It is a huge book, so you're likely to find something useful, but this is due to quantity rather than quality. If you want something akin to a textbook, you won't be disappointed. If you are looking for something to inspire you, or show you how creative programming can be, look elsewhere.

A copy of VB Express is included, making the price more acceptable, but there are other cheaper and better books that offer the same advantage.

SUMMARY

☒ Covers everything

☒ Extremely tedious

SUPPLIER www.amazon.co.uk
ISBN 013225140X
PAGES 1,360

Me" sets the text the button displays. In all cases, the Content property can be set to text or a UIElement such as an Image object.

USING THE PANEL

Sometimes Content consists of multiple objects, and we have to worry about the layout of these. This is dealt with by a Panel object. Usually, we assign a Panel object to the content of a Window and then assign other display objects to the Panel object's layout hierarchy.

Exactly how these objects are arranged depends on the type of Panel object we use. Currently, there are five standard layout Panels: Canvas, which provides absolute x,y positioning; DockPanel, which allows the user to dock elements interactively; Grid, which provides fixed columns and rows for positioning; StackPanel, which arranges objects into a horizontal or vertical line; and WrapPanel, which arranges objects in the same way that items flow down a webpage. The Canvas Panel is closest to traditional Windows forms. The WrapPanel is the one least like a traditional form. To see this in action, enter the following XAML:

```
<WrapPanel>
  <Button
    Height="50"
    Width="200" >
  </Button>
  <Button
```

```
    Height="50"
    Width="200" >
  </Button>
</WrapPanel>
```

If you run the project and resize the form, you will see that the three buttons move to use the available space, just like the text in a webpage.

This particular object hierarchy has been implemented in a rigid and logical way. Because any Content property can be set to another UIElement, this means you can display a button within a button:

```
<Button
  Height="50"
  Width="200" >
  <Button
    Height="20"
    Width="100" >
    Inner Button
  </Button>
</Button>
```

If this isn't mind-boggling enough, you can logically set the content of a button to a Panel and then display lots of other controls within the button. For example:

```
<Button
  Height="200"
  Width="200" >
  <WrapPanel>
    <Button
      Height="20"
      Width="100" >
      Inner Button
    </Button>
    <Button
      Height="20"
      Width="50" >
      Inner Button
    </Button>
  </WrapPanel>
  <Button
    Height="20"
```

```
    Width="40" >
    Inner Button
  </Button>
  <Button
    Height="20"
    Width="100" >
    Inner Button
  </Button>
</WrapPanel>
</Button>
```

The outer button and any of the inner buttons can still be clicked. This example makes for a mad user interface, but in other contexts nested objects are useful.

WORKING WITH GRAPHICS

WPF is also a full graphics system, and you can draw almost anything you want in XAML or in code. For example, to draw a line you would use:

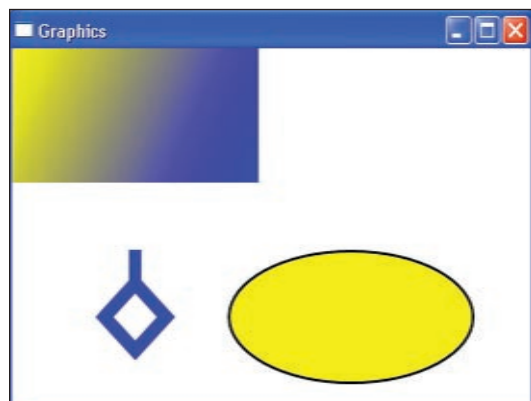
```
<Line
  Stroke="Black"
  StrokeThickness="10"
  X1="10"
  Y1="10"
  X2="75"
  Y2="90"/>
```

An Ellipse is just as easy:

```
<Ellipse
  Height="100"
  Width="200"
  Fill="Yellow"
  Stroke="Black"
  Canvas.Left="175"
  Canvas.Top="50"
  StrokeThickness="2" />
```

You can also draw polylines, a series of straight lines linked together:

```
<Polyline
  Points="25,25 0,50 25,
    75 50,50 25,25 25,0"
  Stroke="Blue"
  StrokeThickness="10"
```



▲ WPF is a complete 2D graphics package


```
Canvas.Left="75"
Canvas.Top="50">
</Polyline>
```

You can also create smooth curves fitted to a set of points using bezier curves, but this is a little more complicated. The documentation explains how it works.

WPF supports a wide range of sophisticated fills including gradient and bitmap fills. For example, to create a linear gradient fill you'd try:

```
<Rectangle Width="200"
Height="100">
  <Rectangle.Fill>
    <LinearGradientBrush
      StartPoint="0,0"
      EndPoint="1,1">
      <GradientStop
        Color="Yellow"
        Offset="0" />
      <GradientStop
        Color="Blue"
        Offset="1" />
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>
```

WORKING WITH ANIMATIONS

One feature of WPF that's well worth trying is animation. The entire graphics system has been built with animation in mind. You can even create animations using nothing but XAML. Here we are going to animate a button that fades in and out when a user is about to click on it, and then jumps some distance when they do. Start a new WPF project, change the layout panel from Grid to Canvas and add a button:

```
<Canvas>
  <Button Name="Button1"
    Canvas.Top="0" Canvas.Left="0"
    Height="100"
    Width="100" >
    Click Me
  </Button>
</Canvas>
```

We have positioned the button at (0,0), the top left-hand corner of the canvas. Notice the way the Canvas property of the button is used to position it. Animations can be created in many ways, but in XAML you have to use a Storyboard object or tag. A storyboard can contain any number of animation objects, which are all activated when it is. Think of an animation object as specifying how a property of the object being animated should change over time. For example:

```
<Storyboard>
  <DoubleAnimation
    Storyboard.TargetName="Button1"
    Storyboard.TargetProperty=
      "Opacity"
    From="1.0"
    To="0.0"
    Duration="0:0:1" />
</Storyboard>
```

This defines a single storyboard with a single animation. It's a 'double' animation because the property being animated is of double precision type, not because it is two animations. There are animation objects to animate points, colour and

so on. If there isn't an animation object for the type of property you want to animate, you can create one. In this case, the animation is applied to Button1's Opacity property. The From To specifies that it should be varied from 1.0 (visible) to 0 (transparent) over a period of one second. This is a simple type of animation, but there are many more sophisticated types, including keyframe and path animations, all working in roughly the same way.

Now we arrive at the problem of when the animation should happen – that is, what triggers it. We could move to code and write event handlers in the usual way, but you can also handle events in XAML. Each control you define in XAML has a Triggers collection. Each trigger specifies an event that invokes it and an action it performs. You can place the Button.Triggers tag straight after the Button tag and start to create a list of triggers:

```
<Button.Triggers>
  <EventTrigger
    RoutedEvent="Button.Mouse
    Enter">
    <EventTrigger.Actions>
```

This defines a single trigger that will occur when the mouse enters the button. The EventTrigger.Actions property is then used to define a list of actions that are performed in response to the event.

We'll start a storyboard running:

```
<BeginStoryboard>
```

We could define the Storyboard object as a named resource and start it running by using its name. However, in this case it's easier to define the storyboard after the BeginStoryboard tag:

```
<Storyboard>
  <DoubleAnimation
    Storyboard.TargetName=
      "Button1"
    Storyboard.TargetProperty=
      "Opacity"
    From="1.0" To="0.0"
    Duration="0:0:1"
    AccelerationRatio="0.5"
    DecelerationRatio="0.5"
    RepeatBehavior="Forever"
    AutoReverse="True" />
  </Storyboard>
</BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
</Button.Triggers>
```

If you run the program at this stage of development, you will see the button fade out repeatedly when you move the mouse over it. The storyboard is now slightly more complicated as we have added Acceleration and Deceleration ratios, which specify how the change speeds up and slows down. We have also made it repeat forever and loop back to its original state.

Once you have grasped the basic idea of animation, you can achieve most effects by experimentation. To make the button move diagonally when the user clicks on it, add the following between the EventTrigger.Actions tag:

```
<EventTrigger RoutedEvent="Button.
Click">
```

```
<EventTrigger.Actions>
  <BeginStoryboard>
    <Storyboard>
      <DoubleAnimation
        Storyboard.TargetName=
          "Button1"
        Storyboard.TargetProperty="
          (Canvas.Left)"
        From="0.0" To="125.0"
        Duration="0:0:1"
        AccelerationRatio="0.5"
        DecelerationRatio="0.5"
        RepeatBehavior="1x"
        AutoReverse="True" />
      <DoubleAnimation
        Storyboard.TargetName="
          Button1"
        Storyboard.TargetProperty="
          (Canvas.Top)"
        From="0.0" To="125.0"
        Duration="0:0:1"
        AccelerationRatio="0.5"
        DecelerationRatio="0.5"
        RepeatBehavior="1x"
        AutoReverse="True" />
    </Storyboard>
  </BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
```

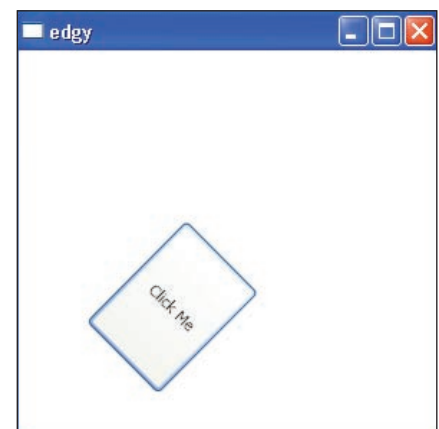
This moves the top and left properties, and repeats the entire action just once.

As you work with XAML, it's easy to forget that you are working with a complete graphics system. All coordinates are in terms of absolute measurements rather than pixels, and if the user interface is displayed on a device with a higher resolution display, the quality increases rather than the size of the text or buttons. There are also transformations that you can apply to any rendered graphic. Add:

```
<Button.RenderTransform>
  <RotateTransform Angle="45" />
</Button.RenderTransform>
```

The button will be drawn at 45°. It will still work as a standard button and even the animations work. It is also possible to animate a transformation.

Once you've seen how WPF works, you should be able to explore most of its facilities with the help of the documentation in the SDK. It's the way ahead that provides a powerful and unified approach to Windows graphics, and you can't afford to ignore it.



▲ Buttons just aren't supposed to behave like this